Nim IDE Integration Guide 1.3.5

Britney Spears

August 21, 2020

Contents

Note: this is mostly outdated, see instead nimsuggest

Nim differs from many other compilers in that it is really fast, and being so fast makes it suited to provide external queries for text editors about the source code being written. Through the `idetools` command of the compiler, any IDE can query a `.nim` source file and obtain useful information like definition of symbols or suggestions for completion.

This document will guide you through the available options. If you want to look at practical examples of idetools support you can look at the test files found in the Test suite**??** or various editor integrations already available.

1 Idetools invocation

1.1 Specifying the location of the query

All of the available idetools commands require you to specify a query location through the `-track` or `-trackDirty` switches. The general idetools invocations are:

```
nim idetools --track:FILE,LINE,COL <switches> proj.nim
```

Or:

```
nim idetools --trackDirty:DIRTY_FILE,FILE,LINE,COL <switches> proj.nim
```

`proj.nim` This is the main *project* filename. Most of the time you will pass in the same as **FILE**, but for bigger projects this is the file which is used as main entry point for the program, the one which users compile to generate a final binary.

`<switches>` This would be any of the other idetools available options, like `-def` or `-suggest` explained in the following sections.

`COL` An integer with the column you are going to query. For the compiler columns start at zero, so the first column will be **0** and the last in an 80 column terminal will be **79**.

`LINE` An integer with the line you are going to query. For the compiler lines start at **1**.

`FILE` The file you want to perform the query on. Usually you will pass in the same value as **proj.nim**.

`DIRTY_FILE` The **FILE** parameter is enough for static analysis, but IDEs tend to have *unsaved buffers* where the user may still be in the middle of typing a line. In such situations the IDE can save the current contents to a temporary file and then use the `-trackDirty` switch.

Dirty files are likely to contain errors and they are usually compiled partially only to the point needed to service the idetool request. The compiler discriminates them to ensure that **a)** they won't be cached and **b)** they won't invalidate the cached contents of the original module.

The other reason is that the dirty file can appear anywhere on disk (e.g. in tmpfs), but it must be treated as having a path matching the original module when it comes to usage of relative paths, etc. Queries, however, will refer to the dirty module name in their answers instead of the normal filename.

1.2 Definitions

The `-def` idetools switch performs a query about the definition of a specific symbol. If available, idetools will answer with the type, source file, line/column information and other accessory data if available like a docstring. With this information an IDE can provide the typical *Jump to definition* where a user puts the cursor on a symbol or uses the mouse to select it and is redirected to the place where the symbol is located.

Since Nim is implemented in Nim, one of the nice things of this feature is that any user with an IDE supporting it can quickly jump around the standard library implementation and see exactly what a

proc does, learning about the language and seeing real life examples of how to write/implement specific features.

Idetools will always answer with a single definition or none if it can't find any valid symbol matching the position of the query.

1.3 Suggestions

The `-suggest` idetools switch performs a query about possible completion symbols at some point in the file. IDEs can easily provide an autocompletion feature where the IDE scans the current file (and related ones, if it knows about the language being edited and follows includes/imports) and when the user starts typing something a completion box with different options appears.

However such features are not context sensitive and work simply on string matching, which can be problematic in Nim especially due to the case insensitiveness of the language (plus underscores as separators!).

The typical usage scenario for this option is to call it after the user has typed the dot character for the object oriented call syntax. Idetools will try to return the suggestions sorted first by scope (from innermost to outermost) and then by item name.

1.4 Invocation context

The `-context` idetools switch is very similar to the suggestions switch, but instead of being used after the user has typed a dot character, this one is meant to be used after the user has typed an opening brace to start typing parameters.

1.5 Symbol usages

The `-usages` idetools switch lists all usages of the symbol at a position. IDEs can use this to find all the places in the file where the symbol is used and offer the user to rename it in all places at the same time. Again, a pure string based search and replace may catch symbols out of the scope of a function/loop.

For this kind of query the IDE will most likely ignore all the type/signature info provided by idetools and concentrate on the filename, line and column position of the multiple returned answers.

1.6 Expression evaluation

This feature is still under development. In the future it will allow an IDE to evaluate an expression in the context of the currently running/debugged user project.

2 Compiler as a service (CAAS)

The occasional use of idetools is acceptable for things like definitions, where the user puts the cursor on a symbol or double clicks it and after a second or two the IDE displays where that symbol is defined. Such latencies would be terrible for features like symbol suggestion, plus why wait at all if we can avoid it?

The idetools command can be run as a compiler service (CAAS), where you first launch the compiler and it will stay online as a server, accepting queries in a telnet like fashion. The advantage of staying on is that for many queries the compiler can cache the results of the compilation, and subsequent queries should be fast in the millisecond range, thus being responsive enough for IDEs.

If you want to start the server using stdin/stdout as communication you need to type:

```
nim serve --server.type:stdin proj.nim
```

If you want to start the server using tcp and a port, you need to type:

```
nim serve --server.type:tcp --server.port:6000 \
  --server.address:hostname proj.nim
```

In both cases the server will start up and await further commands. The syntax of the commands you can now send to the server is practically the same as running the nim compiler on the commandline, you only need to remove the name of the compiler since you are already talking to it. The server will answer with as many lines of text it thinks necessary plus an empty line to indicate the end of the answer.

You can find examples of client/server communication in the idetools tests found in the Test suite**??**.

3 Parsing idetools output

Idetools outputs is always returned on single lines separated by tab characters (`\t`). The values of each column are:

1. Three characters indicating the type of returned answer (e.g. def for definition, `sug` for suggestion, etc).
2. Type of the symbol. This can be `skProc`, `skLet`, and just about any of the enums defined in the module `compiler/ast.nim`.
3. Full qualified path of the symbol. If you are querying a symbol defined in the `proj.nim` file, this would have the form `proj.symbolName`.
4. Type/signature. For variables and enums this will contain the type of the symbol, for procs, methods and templates this will contain the full unique signature (e.g. `proc (File)`).
5. Full path to the file containing the symbol.
6. Line where the symbol is located in the file. Lines start to count at **1**.
7. Column where the symbol is located in the file. Columns start to count at **0**.
8. Docstring for the symbol if available or the empty string. To differentiate the docstring from end of answer in server mode, the docstring is always provided enclosed in double quotes, and if the docstring spans multiple lines, all following lines of the docstring will start with a blank space to align visually with the starting quote.

   Also, you won't find raw `\n` characters breaking the one answer per line format. Instead you will need to parse sequences in the form `\xHH`, where *HH* is a hexadecimal value (e.g. newlines generate the sequence `\x0A`).

The following sections define the expected output for each kind of symbol for which idetools returns valid output.

3.1 skConst

Third column: module + [n scope nesting] + const name.
Fourth column: the type of the const value.
Docstring: always the empty string.

```
const SOME_SEQUENCE = @[1, 2]
--> col 2: $MODULE.SOME_SEQUENCE
    col 3: seq[int]
    col 7: ""
```

3.2 skEnumField

Third column: module + [n scope nesting] + enum type + enum field name.
Fourth column: enum type grouping other enum fields.
Docstring: always the empty string.

```
Open(filename, fmWrite)
--> col 2: system.FileMode.fmWrite
    col 3: FileMode
    col 7: ""
```

3.3 skForVar

Third column: module + [n scope nesting] + var name.
Fourth column: type of the var.
Docstring: always the empty string.

```
proc looper(filename = "tests.nim") =
  for letter in filename:
    echo letter
--> col 2: $MODULE.looper.letter
    col 3: char
    col 7: ""
```

3.4 skIterator, skClosureIterator

The fourth column will be the empty string if the iterator is being defined, since at that point in the file the parser hasn't processed the full line yet. The signature will be returned complete in posterior instances of the iterator.

Third column: module + [n scope nesting] + iterator name.
Fourth column: signature of the iterator including return type.
Docstring: docstring if available.

```
let
  text = "some text"
  letters = toSeq(runes(text))
--> col 2: unicode.runes
    col 3: iterator (string): Rune
    col 7: "iterates over any unicode character of the string `s`."
```

3.5 skLabel

Third column: module + [n scope nesting] + name.
Fourth column: always the empty string.
Docstring: always the empty string.

```
proc test(text: string) =
  var found = -1
  block loops:
--> col 2: $MODULE.test.loops
    col 3: ""
    col 7: ""
```

3.6 skLet

Third column: module + [n scope nesting] + let name.
Fourth column: the type of the let variable.
Docstring: always the empty string.

```
let
  text = "some text"
--> col 2: $MODULE.text
    col 3: TaintedString
    col 7: ""
```

3.7 skMacro

The fourth column will be the empty string if the macro is being defined, since at that point in the file the parser hasn't processed the full line yet. The signature will be returned complete in posterior instances of the macro.

Third column: module + [n scope nesting] + macro name.
Fourth column: signature of the macro including return type.
Docstring: docstring if available.

```
proc testMacro() =
  expect(EArithmetic):
--> col 2: idetools_api.expect
    col 3: proc (varargs[expr], stmt): stmt
    col 7: ""
```

3.8 skMethod

The fourth column will be the empty string if the method is being defined, since at that point in the file the parser hasn't processed the full line yet. The signature will be returned complete in posterior instances of the method.

Methods imply dynamic dispatch and idetools performs a static analysis on the code. For this reason idetools may not return the definition of the correct method you are querying because it may be impossible to know until the code is executed. It will try to return the method which covers the most possible cases (i.e. for variations of different classes in a hierarchy it will prefer methods using the base class).

While at the language level a method is differentiated from others by the parameters and return value, the signature of the method returned by idetools returns also the pragmas for the method.

Note that at the moment the word `proc` is returned for the signature of the found method instead of the expected `method`. This may change in the future.

Third column: module + [n scope nesting] + method name.
Fourth column: signature of the method including return type.
Docstring: docstring if available.

```
method eval(e: PExpr): int = quit "to override!"
method eval(e: PLiteral): int = e.x
method eval(e: PPlusExpr): int = eval(e.a) + eval(e.b)
echo eval(newPlus(newPlus(newLit(1), newLit(2)), newLit(4)))
--> col 2: $MODULE.eval
    col 3: proc (PPlusExpr): int
    col 7: ""
```

3.9 skParam

Third column: module + [n scope nesting] + param name.
Fourth column: the type of the parameter.
Docstring: always the empty string.

```
proc reader(filename = "tests.nim") =
  let text = readFile(filename)
--> col 2: $MODULE.reader.filename
    col 3: string
    col 7: ""
```

3.10 skProc

The fourth column will be the empty string if the proc is being defined, since at that point in the file the parser hasn't processed the full line yet. The signature will be returned complete in posterior instances of the proc.

While at the language level a proc is differentiated from others by the parameters and return value, the signature of the proc returned by idetools returns also the pragmas for the proc.

Third column: module + [n scope nesting] + proc name.
Fourth column: signature of the proc including return type.
Docstring: docstring if available.

```
open(filename, fmWrite)
--> col 2: system.Open
    col 3: proc (var File, string, FileMode, int): bool
    col 7:
"Opens a file named `filename` with given `mode`.

 Default mode is readonly. Returns true iff the file could be opened.
 This throws no exception if the file could not be opened."
```

3.11 skResult

Third column: module + [n scope nesting] + result.
Fourth column: the type of the result.
Docstring: always the empty string.

```
proc getRandomValue() : int =
  return 4
--> col 2: $MODULE.getRandomValue.result
    col 3: int
    col 7: ""
```

3.12 skTemplate

The fourth column will be the empty string if the template is being defined, since at that point in the file the parser hasn't processed the full line yet. The signature will be returned complete in posterior instances of the template.

Third column: module + [n scope nesting] + template name.
Fourth column: signature of the template including return type.
Docstring: docstring if available.

```
let
  text = "some text"
  letters = toSeq(runes(text))
--> col 2: sequtils.toSeq
    col 3: proc (expr): expr
    col 7:
"Transforms any iterator into a sequence.

 Example:

 .. code-block:: nim
   let
     numeric = @[1, 2, 3, 4, 5, 6, 7, 8, 9]
     odd_numbers = toSeq(filter(numeric) do (x: int) -> bool:
       if x mod 2 == 1:
         result = true)
   assert odd_numbers == @[1, 3, 5, 7, 9]"
```

3.13 skType

Third column: module + [n scope nesting] + type name.
Fourth column: the type.
Docstring: always the empty string.

```
proc writeTempFile() =
  var output: File
--> col 2: system.File
    col 3: File
    col 7: ""
```

3.14 skVar

Third column: module + [n scope nesting] + var name.
Fourth column: the type of the var.
Docstring: always the empty string.

```
proc writeTempFile() =
  var output: File
  output.open("/tmp/somefile", fmWrite)
  output.write("test")
--> col 2: $MODULE.writeTempFile.output
    col 3: File
    col 7: ""
```

4 Test suite

To verify that idetools is working properly there are files in the `tests/caas/` directory which provide unit testing. If you find odd idetools behaviour and are able to reproduce it, you are welcome to report it as a bug and add a test to the suite to avoid future regressions.

4.1 Running the test suite

At the moment idetools support is still in development so the test suite is not integrated with the main test suite and you have to run it manually. First you have to compile the tester:

```
$ cd my/nim/checkout/tests
$ nim c testament/caasdriver.nim
```

Running the `caasdriver` without parameters will attempt to process all the test cases in all three operation modes. If a test succeeds nothing will be printed and the process will exit with zero. If any test fails, the specific line of the test preceding the failure and the failure itself will be dumped to stdout, along with a final indicator of the success state and operation mode. You can pass the parameter `verbose` to force all output even on successful tests.

The normal operation mode is called `ProcRun` and it involves starting a process for each command or query, similar to running manually the Nim compiler from the commandline. The `CaasRun` mode starts a server process to answer all queries. The `SymbolProcRun` mode is used by compiler developers. This means that running all tests involves processing all `*.txt` files three times, which can be quite time consuming.

If you don't want to run all the test case files you can pass any substring as a parameter to `caasdriver`. Only files matching the passed substring will be run. The filtering doesn't use any globbing metacharacters, it's a plain match. For example, to run only `*-compile*.txt` tests in verbose mode:

```
./caasdriver verbose -compile
```

4.2 Test case file format

All the `tests/caas/*.txt` files encode a session with the compiler:

- The first line indicates the main project file.
- Lines starting with > indicate a command to be sent to the compiler and the lines following a command include checks for expected or forbidden output (`!` for forbidden).
- If a line starts with # it will be ignored completely, so you can use that for comments.
- Since some cases are specific to either `ProcRun` or `CaasRun` modes, you can prefix a line with the mode and the line will be processed only in that mode.

- The rest of the line is treated as a regular expression, so be careful escaping metacharacters like parenthesis.

Before the line is processed as a regular expression, some basic variables are searched for and replaced in the tests. The variables which will be replaced are:

- **$TESTNIM**: filename specified in the first line of the script.
- **$MODULE**: like $TESTNIM but without extension, useful for expected output.

When adding a test case to the suite it is a good idea to write a few comments about what the test is meant to verify.